目录

摘要

PKI 是现代电子通讯在非对称私钥加密通讯环境下所必须的一种基本通讯设施，为通讯双方的公钥查询提供了安全的认证保障。而通过区块链构建 PKI 的技术，解决了现今通讯环境下 PKI 中心网络实体易受攻击和遭受信息伪造的问题，本文总结了区块链下的 PKI 在理论和实际应用情况下的可用性和安全性，并在以太坊私有链环境下搭建了可满足基本功能的 PKI 合约，与现有的 PKI 对比，探索了真实使用中该 PKI 体系可能出现的问题及改进意见。

关键词：区块链；公钥基础设施；以太坊；智能合约

Abstract

PKI is a basic communication facility for modern electronic communications under the asymmetric private key encrypted communication environment. It provides secure authentication for public key queries of both parties. The technology of constructing PKI through blockchain solves the problem that PKI-centric network entities are vulnerable to attack and information forgery under the current communication environment. This article summarizes the availability and security of PKI under blockchain theory and practical applications. Sex, and built a PKI contract that meets basic functions in the private chain environment of Ethereum. Compared with the existing PKI, it explores possible problems and improvements in the PKI system in real use.

Keywords: Block chain; PKI; Ethereum; Smart contract

前言

区块链是一种以去中心化的分布式计算框架为技术核心的数据库展示方案，其独特的去中心化、不可修改、公众维护、集体安全等特性，使得它在现实生活中的应用愈加广泛[1]。而一个安全的 PKI(Public Key Infrastructure) 系统，恰恰需要区块链的这些卓越特性，来保证其用户在公钥的存储、更新、获取等操作安全可信。因此，本文详细介绍了在区块链下构建的 PKI 模型，分析了其理论上与传统模型在运作和安全性的区别，并探讨了改进意见；随后通过以太坊 GETH 环境下的私有链，搭建一个可以满足基本 PKI 要求的操作过程，为区块链下的 PKI 用户化实现提供了参考。

结构及安全性介绍

简单来说，基于区块链的 PKI 系统就是一个建立在区块链上的公钥基础设施，它需要具有 PKI 体系对公钥一系列操作的功能，同时又需要通过区块链来消除传统 PKI 中心化的问题并保证其公钥的安全性。故在这一章节，我们将在章节 1 中介绍传统区块链的构成方式和运行过程中的安全保障；在章节 2 中介绍传统 PKI 的基本概念和核心功能。

1. 区块链基本概念及安全机理

区块链在近年来作为比特币等数字模拟货币的技术支撑，成为了研究的热点，但去除经济等人为因素，其作为分布式交互环境的一个新型结构，在我们的生活中正扮演着重要的作用，尤其在金融领域，区块链以去中心化记账的代表而应用广泛[2]。

1.1 区块链基本概念

顾名思义，区块链的基础就是两个方面：区域区块化和链式结构。区块的特质是指账本中的交易数据通过区块的形式被记录下来，全系统共享并且共同

进行维护；而链式结构中，不同区块由时间戳进行前后的区分，各个区块之间又通过链式结构进行连接。很显然，区块链是以分布式记账系统的形式存在的。但与此同时，就会存在分布式网络应用的典型问题：共识问题和拜占庭将军问题。而区块链的出现可以认为是以在 2008 年中本聪发表的论文《Bitcion: A Peer-to-Peer Electronic Cash System》[4]为代表，其中关于比特币的构想，解决了区块链应用的难题。遵循比特币虚拟货币、分布式区块链的基本思想，现如今区块链已经有很多变种，其中最为著名的就是开源以太坊[5]，以太坊是 2013 年 Vitalik Buterin 提出的分布式应用平台，其应用十分广泛，下文也将以以太坊为代表描述其安全机理。

1.2 区块链安全机理

主流区块链在构成的过程中，大多是通过 hash 函数对每个区块内的数据进行检验来保证其完整性的。并且每个区块通过添加上一个区块的 hash 值来进行链接，使之成为一个链式结构。与此同时，hash 函数的抗碰撞性也使得修改已入链的区块信息是多项式不可能的[6]。而区块链接到区块链主链的过程中，每个区块的时间戳和共识算法使得同时出现两个支链的情况可以被避免，从而保证所有节点接受的信息是一致的。当区块链接到主链上后，区块链的分布式记账结构使得每一个参与节点都保存有该主链的全部信息，就算敌手通过了 hash 碰撞检验，妄图修改链中已有信息也需要克服工作量证明而推翻系统一半以上的算力。在运行中的大型区块链中，这是近乎不可能的事情。其安全机制的具体参照如下：

Hash 算法 hash 是密码学算法的重要部分，它的作用是将任意长度的明文通过单向函数转化成固定字长，方便进行文本处理并提供安全的 MAC。区块链技术大范围采用了 hash 算法，作为密码加密的预处理和区块之间的链接、同时也给 Mapping 函数的 DHT（distinction hash table）提供了简便的查找方式。区块链中信息在上传的过程中，会在所在区块生成该信息的 hash 函数，同时在每个区块中含有上个区块的 hash 值，也计算入该区块的 Hash 过程。系统中的每个用户都可以在任意时段对开放的区块信息进行 hash 检测，Hash 算法的抗碰撞性完美的保障了区块链信息的完整性和链式结构的不可否认性。我们用到的以太

坊 GETH 命令环境采用的是 SHA-3-256 算法，输出为 32 字节，安全指数为 168.

共识过程 以太坊的共识过程是通过对备选链条的既有工作量总额进行筛选完成的，对于短时间内出现的不同分叉同时指向前一个区块，各节点总是选择已有工作量证明最大的链作为主链，其他的支链将会被抛弃[7]。我们知道，区块链的运行依靠着数字模拟货币，而模拟货币的重要问题是重复支付问题，需要避免货币拥有者使用同一笔钱进行两次支付。区块链提供的共识算法一方面通过时间戳进行排序，另一方面通过共识算法选择累积工作量最大的链条（一般比特币中是需要通过 6 个区块周期确定主链），从而避免出现两笔交易同时发生的状况。

工作量证明（Proof of Work） 一个交易在被整个系统承认之前需要完成检验，而工作量证明提供的就是这么一种验证过程。以太坊的工作量证明是通过大量的计算去寻找一个动态的区块 hash 值完成的，这个计算的复杂度由根据当前链中工作节点的数量而动态调整的 hash 值通过门限决定，使得完成一个区块的验证能够维持在 15s[8]。而工作量证明的完成是依靠挖矿，通过计算资源完成一个包含有效交易的区块创建的矿工拥有区块的记账权。POW 的存在使得伪造一个可通过检验的区块变得十分困难，同时共识算法也使得推翻既有节点要求伪造者拥有总结点至少一半的算力，从而解决了拜占庭将军问题，即分布系统下各节点的信任问题。

2. PKI 的基本概念及核心功能

PKI 是公钥基础设施（Public Key Infrastructure）的简称，传统 PKI 是由认证中心将用户的公钥与用户认证信息结合认证中心的签名，构成数字认证证书，并以证书的方式提供用户公钥的认证服务。PKI 的通信方式采用标准 C/S 模式，沟通非对称签名标准并颁发证书，在这个过程中，认证中心的权威性是采纳用户公钥的唯一可信源。

2.1 PKI 基本构成

根据 ITU 发布的 X.509 标准，PKI 要求能够支持公开密钥的管理并能支持认证、加密、完整性和可追究性服务，因此构成一个完整的 PKI 系统，需要包

括以下五个主要方面[9]：

1．认证机构，可信的证书签发机构；

2. 证书库，用于存放可供公众查询的、其他经过认证机构认证并签名的证书；

3. 密钥恢复和备份系统，对用户的公钥密钥进行备份，并提供恢复服务；

4. 证书撤销处理系统，对于作废的证书及时实现终止使用，撤销证书的CRL 列表；

5. PKI 应用接口，保证与 PKI 构成交互的网络连接安全可靠。

而上述的 PKI 构成要素同时也包含着 PKI 需要提供给用户的服务。在 PKI 系统中每个用户 u 拥有自己的密钥对 keypair（sk,pk），该密钥对可以实现非对称加密过程[10]，算法如下：

- $\text{keygen}(1^k) \rightarrow keypair(sk,pk)$,给定安全参数$k$，生成密钥对$sk$、$pk$。
- $sig_{sk}(m) \rightarrow s$，使用密钥$sk$对$m$完成签名操作，得到s。
- $ver_{pk}(m，s) \rightarrow \{0,1\}$，使用$pk$对(s,m)进行验证操作，验证成功返回结果 1，不然返回结果 0.

因此，作为公钥加密体系的基础设置，PKI 最基本的工作就是要记录用户提供的pk，并在其他用户查询该pk时返回该pk值。因此 PKI 所提供的服务在用户方面是：用户注册其pk、核对通过后颁发认证中心的证书 CA、用户可以更新其提交的pk、用户可注销已提交的 pk；而在使用用户方面是：给定用户id可查询id对应的pk。

其实现过程为

- $(id,pk) \rightarrow CA_{skca}(id,pk,r)$，用户id提交可用$pk$，认证中心验证该$pk$的正确性后颁发证书 CA，证书 CA 是认证中心通过自己私钥$skca$对用户pk和该用户id相关的证书信息的签名；
- $id \rightarrow CA_{skca}(id,pk,r)$，使用者向认证中心发送对id的,$pk$查询请求，认证中心返回该$id$的证书 CA；
- $CA_{skca}(id,pk,r) \rightarrow (\text{id,pk})$，使用者得到证书 CA，通过已知的认证中心公钥pkca对签名进行认证得到所求id的pk；
- $(id,update,pkold,pknew) \rightarrow CA_{new}$，用户提交更新pk请求，并告知旧

$,pk$和新$,pk$，认证中心核对信息通过后，颁发新证书CA_{new}，并更新自己存储的 CA；

- $(id,revoke,pk) \rightarrow \{0,1\}$，用户提交注销$pk$请求，认证中心核对信息后，通过核对后，如注销成功则返回 1，否则返回信息 0.

2.2 PKI 的安全性依赖

PKI 系统在使用过程中的安全性完全依赖于安全的非对称签名算法，非对称签名算法的安全性直接决定了包括用户使用过程的私钥安全和证书 CA 是否可被伪造的安全性。而对于 PKI 体系的安全性，依托于认证中心的层次模型。各级 CA 认证机构构成了层次模型中的信任链，彼此之间的链接通过上级向下级颁发证书来实现。在 X.509 标准中详细阐明了其运行机理：这种认证体系将证书分成，各证书都有上级 CA 的数字签名，当用户查看对方证书是否可信时，首先看其上级 CA 的签名，上级 CA 真实可信，则继续查看其对应的而上级 CA 的公钥来解密其数字签名，以此类推，一直查到证书颁发者的名字与证书的主体名字相同时为止，也就是查到根证书与自签名证书为止。

但是这样的层次结构会引发出这样的问题:根证书或自签名证书的安全性如何保障？在传统 PKI 体系中，我们不得不去相信至少一个第三方机构的可信度，在上述的层次模型中，我们至少要相信根证书的颁发机构，这是一切 PKI 体系运转的前提。当然这就引发了一系列质疑和一直困扰我们的隐患：如何使人相信证书颁发机构？证书颁发机构如何规避受攻击风险？证书颁发机构受到攻击时该如何保障 PKI 的正常运转？也正因此，2016 年 MIT 的 Conner Fromknecht,Dragos Velicanu 和 Sophia Yakoubov 提出了依靠区块链的去中心化结构运作的 PKI 体系——certcoin[11]，使得 PKI 体系中必须信任某一第三方机构这一问题得到较好的解决。

基于区块链的 PKI 系统 certcoin

1. 体系介绍

基于区块链的 PKI 系统构想为通过区块链技术，将某一固定范围区域链声明为 PKI：节点用户发布的pk以及一些相关信息经验证后以区块的方式在区块链上公布作为公钥的注册操作；其后，通过验证的发布者可以通过提交新的公钥pk_{new}和注销在链中新建区块覆盖原有pk信息；节点内任意用户均可对链中已公布pk进行查询操作。

注册操作 注册拥有者向 PKI 提交命名信息为 name 的注册信息，PKI 通过验证后将该消息公布在自己所拥有的区块链中

- (owner, name, skn, skf) →

  (id, name, register, pkn, pkf, valuen[$sig_{pkn}(id)$], valuef[$sig_{pkf}(id)$])，

  给定name，注册拥有者owner生成并发送注册信息，value为验证签名，通过公布的公钥所对应的私钥对skn、skf对id的签名来实现；

- $(\text{id, name, register, pkn, pkf, valuen}[sig_{skn}(id)], \text{valuef}[sig_{skf}(id)]) \rightarrow$

  {ture, false}，

  PKI 接受验证信息，首先查询链中是否具有可用且相同的name信息，没有则对value进行签名验证，验证通过则公布该name的pk及相关信息；

更新操作 任何用户可以向 PKI 提交已存在的命名信息为name的更新消息，PKI 通过对旧签名的验证来确认更新信息是否可信，验证通过即在所拥有区块链中公布更新消息并对该name进行更新标记

- (name, sko) →

  (id, name, update, keytype, pkw, pko, valuew[$sig_{skw}(id)$], valueo[$sig_{sko}(id + pkw)$])，

  给定name，用户生成更新信息，valuew, valueo为验证签名，通过公布的公钥对应的私钥对skw对id和sko对id + pkw的签名来实现；其中公钥类型 pkf 比 pkn 具有更高的更新权限；

- (id, name, update. keytype, pkw, pko, valuew[$sig_{skw}(id)$], valueo[$sig_{sko}(id + pkw)$]) → {ture, false}，

  PKI 接受验证信息，首先查询链中是否具有可用且相同的name信息，有则对value进行签名验证，验证通过则公布该name更新后的pkw及相关信息；

注销操作 任何用户可以向 PKI 提交已存在的命名信息为name的注销消息，PKI 通过对签名的验证来确认注销信息是否可信，验证通过即在所拥有区块链中公布更新消息并对该name进行注销操作并标记

- (name, sk) → (id, name, revoke, valuen[$sig_{skn}(id + revoke)$], valuef[$sig_{skf}(id + revoke)$])，

  给定name，用户生成更新信息，valuen, valuef为验证签名，通过公布的公钥对应的私钥对skn对id + revoke和skf对id + revoke的签名来实现；
- (id, name, revoke, valuen[$sig_{skn}(id + revoke)$], valuef[$sig_{skf}(id + revoke)$]) → {ture, false}，

  PKI 接受验证信息，首先查询链中是否具有可用且相同的name信息，有则对value进行签名验证，验证通过则返回ture并注销name的pk及相关信息；

查询操作 任何用户可向 PKI 提交命名信息为 name 的查询信息，PKI 收到请求后将在自己所拥有的区块链中查询对应 pk 并返回

- (id, name) → (id, name, request, pktype)，

  给定name 或 id，使用者生成并发送查询信息，pktype为请求的公钥类型；
- (id, name, request, pktype) → (pk)，

  PKI 接受查询请求，首先查询链中是否具有可用的name信息，有则返回对应对pktype的pk值；

2. 结构对比

较之传统 PKI 系统，区块链 PKI 具有着天然的优势。首先，区块链透明化的

分布式记账规则使得用户公布在区块链上的公钥pk可以被公共清楚地获取，pk的注册、更新、注销等消息也均会详细地记录在区块链 MK 树上，可供查询，针对 PK 信息的每一次更新和注销等操作也是对每一个参与节点开放可查的；其次，区块链由工作量证明和 hash 算法所提供的天然的不可更改性，使得用户公布的 pk 很难被攻击者篡改，有效的抵抗了存在性攻击保证了pk的可信度；最后，区块链的分布式规则，完美取消了公众对传统 PKI 系统中认证中心容易受到攻击的顾虑，因为区块链中的信息是同步在区块链的每一个节点中的，攻击者的每一个可行的攻击都可以在pk被修改后被系统推选出的公共节点信息恢复，何况，攻击者的攻击难度是被扩大到了系统内至少一半节点容量$\mathrm{N} = |V|$的规模大小，显然，当区域节点很多的情况下，正常的攻击在 polynomial 条件下是不可能存在。

但同时区块链环境下的 PKI 也会存在一些问题。最明显的就是区块链节点信息存储量问题：区块链解决了取消中心节点的难题，但代价是系统中每个或大部分节点都需要保存公共链中的全部数据，不仅占据了大量的空间资源，还增加了节点对链内信息搜查的难度；其次是区块链的不可更改性：相较于传统数据存储库对数据可以进行增加、删除、修改、查询四种操作，但区块链的数据操作因为区块链信息一旦公布就无法修改的特性而没有修改和删除操作，故对区块链 PKI 中的 PK 数据修改操作只有更新和注销两种，更新是通过再次声明的方式对原有数据进行覆盖，因此查询过程就要求用户需要通过时间戳对每一个声明区块进行查找操作，无疑再次增加了对信息查询的难度。

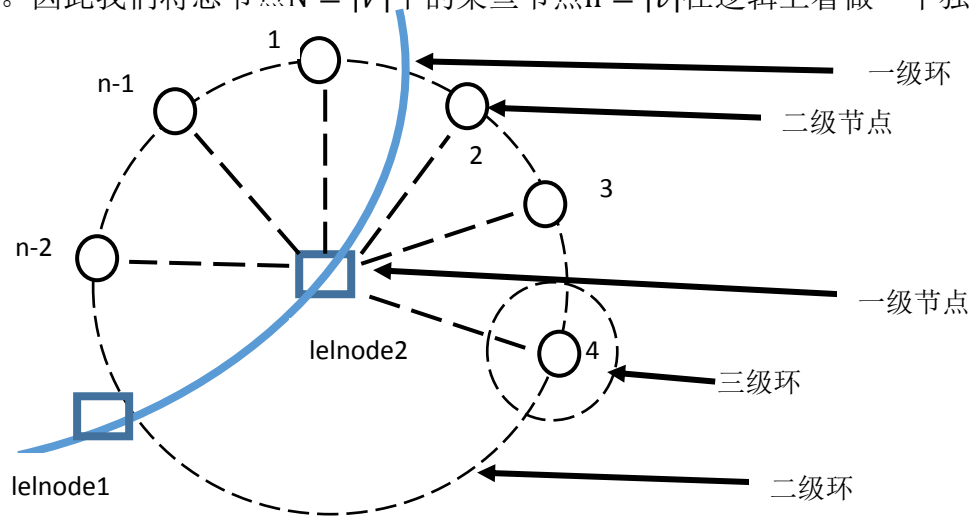
3. 改进建议

针对上文所说的，区块链所固有的空间和时间上的效率问题，我们根据区块链节点的 P2P 特性，提出了区域性多级节点的存储模型，将降低$\frac{\mathrm{N}-\log(\mathrm{N})}{N}$个节点（大约 97%）的存储量，其中 98%的节点只需要存储所在层数 n 的2^{n-1}个节点$(1/2)^n$的存储量，按现有以太坊 10G 的存储量，有效减少50%~99.99%的存储空间。而在时间效率上采用通用的 DHT 算法，可控制在$\mathrm{O}(\log(\mathrm{N}))$个通讯时间内完成查询工作。我们的具体方案如下。

3.1 空间占用问题

对于空间占用大的问题，我们可以采用 P2P(Peer-to-Peer)技术解决此问题，因此借助 P2P 节点的经典 Chord 环算法[12]和 Kademlia 算法[13]，我提出了区域性多级节点 RML(Regional Multi-level Technology) 构想完成节点存储：

对于区块链上的每个节点，理论上是要求包含有整个区块链的全部区块信息才能参与区块链的操作，随着区块链长度的不断增长，无疑会不断地增加节点的存储空间。因此我们将总节点$N = |V|$中的某些节点$n = |v|$在逻辑上看做一个独



pic 1：RML 存储结构

立的区域，设置其为二级节点环 roll2，如图 1 所示。

设置过程

- 参与计算的区块链节点为{lelnode1、lelnode2 …… lelnode(2^N)}，节点按通讯区域（或 IP）划分为若干组，每组数量 n 满足条件$n < N$，选出组中存储空间最大、算力最高的节点作为该组与区块链的链接节点（一级节点）lelnode1，该节点同步存储区块链中的全部信息；
- 分组中n个节点形成逻辑上的放射环roll2结构，除中心节点lelnode1以外，其余$n-1$个节点成为彼此之间保持环结构通讯的二级节点，节点为{lel2node1、lel2node2 …… lel2node$(n-1)$}，并与中心节点lelnode1保持放射型通讯；
- 被划分为lel2node的节点按照 Chord 结构进行主副节点划分，每个节点

只保留自己查询过的区块信息，而其中的$(m \leq \log(n-1))$个节点成为主节点{lel2node1、lel2node2 …… lel2nodem}，这些主节点lel2nodei还需存储与上个主节点间$[lel2node(pre(i-1)), lel2nodei]$的所有副节点存储的信息（信息同步频率视所管理的副节点的查询情况设定），其中对于主节点lel2nodei，其管理的副节点有2^i个，作为通讯的三级节点，与其主节点按步骤 2 继续划分；

- 环roll(x)中的主节点划分为 2 个时，划分结束。

检索方式

发出信息检索需求的节点称为源节点，需查询的区块信息称为目标信息score，源节点对目标节点的检索方式如下：

- 源节点 a 先查询自己存储的数据中是否含有目标信息score，有则查询结束返回结果；没有则向所属主节点发布迭代查询请求；
- 主节点接到子节点的查询请求，查询自己存储数据中是否含有目标信息score，有则查询结束并向源节点 a 返回结果；没有则向自己所属的上级节点 b 发布迭代查询请求；
- 重复步骤 2，直至到达一级节点lelnode1，如果自己有目标信息score，则返回结果，没有则返回查询失败回应；
- 源节点 a 收到查询失败回应则返回查询失败；收到查询成功回应则返回查询结果。

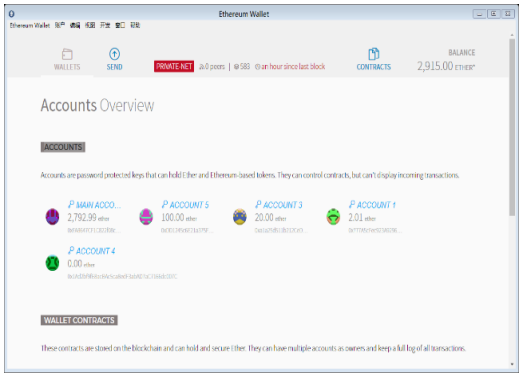
RML 算法使得每个节点存储信息$info \leq INFO$，$INFO(2^N)$ 为全链信息；其中只有第roll1的主节点逻辑上仍然是在区块节点内，且存储全部节点信息；其余节点，对于第 x 级环的主节点来说，其存储空间最大为$2^{n-x}(n = \log N)$，远小于INFO；且对于查询中的流量开销问题，查询越活跃的区域，其所属的主节点对所请求区块的缓存信息越多，会逐渐降低向所属上一节点乃至一级节点发送查询请求的频率；除此之外，分级始终为有限，按以太坊 10G 的节点存储量，最大分级不会超过 4 级，通讯效率仍然保持高效；同时因为分组是按通讯效率或者 IP 进行的划分，可以将通讯流量有效控制在区域通讯的内部，进一步降低了通讯时间和传输效率。

3.2 查询效率问题

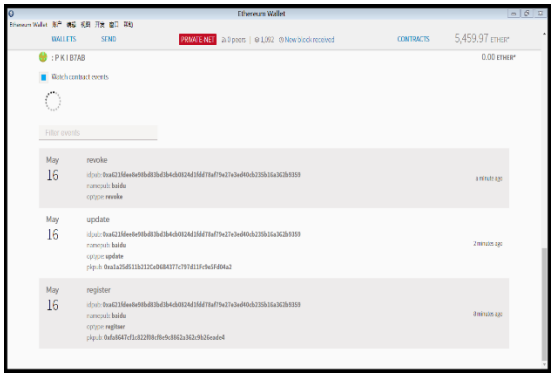
对于解决节点信息的查询效率问题,通常采用散列表 DHT(distributed hash table）技术[14]进行解决。散列表是一种用于存储映射和集合的数据结构，其存储和检索数据可以在时间复杂度O（1）内完成，一个强健的的分布式散列表，可以有效地完成许多分布式算法的需求。在区块链中，每一个区块都有自己在该区块中独一无二的地址(address)编号。以太坊中的地址分为两种：用户地址和合约地址，但是地址编号和区块在链中位置并无必然关系，每次查询需要逆 MK 树进行逆向查询,在总区块数为$N = |V|$的规模下,查询时间复杂度为log(N) [15]。但通过在节点中建立一个随 pk 信息增长而变化的列表的方式，以可用 pk 的用户的 name 或 id 信息为键值构建映射，组成散列表，使该列表存储区块的位置信息。这样可将查询的时间复杂度降低为O（1），极大提升查询效率。

应用

本文通过以太坊私有链采用 solidity 语言部署具有 PKI 交互功能的智能合约。其中以太坊（Ethereum）是第一个也是目前全球最活跃的区块链 2.0 公有链[16]，在区块链体系中极具代表性，它是有 2013 年末 Vitalik Buterin 提出的一个开源的有智能合约功能的公共区块链平台，通过专用加密货币以太币支持以太坊内的交互和信息处理工作[17]。其采用的智能合约由 solidity 语言完成合约编程过程，进而由 EVM（Ethereum Virtual Machine）语言完成编译。合约部署完成后，交易方的交互均是在 GETH 完成。而区块链的私有链是一条个人搭建



pic2.Ethereum wallet



pic3.运行状态

的、不对外公开的区块链，其构成原理与公有链是一致的，但具有只有拥有者具有写入权限、高度隐私的优点，方便我们对 PKI 的性能进行相关测试。

如 pic2 显示了我们的私有链的一些基本信息，可以看到该私有链结构完整、功能完善，共 5 个可用的用户节点并已分配好以太币以用于交易，且已对示例代码进行试验，显示运行通畅。我们经过挖矿确认，将已经编译好的合约部署在私有链中，按要求已经成功生成签名的用户向私有链 PKI 地址（即部署的 PKI 合约地址）提交拥有的 pk，依次完成注册、更改操作，并使用用户地址向 PKI 发起查询操作，最后对 pk 进行注销。其演示结果如 pic3 所示。可以看到各项操作均成功，程序运行流畅。

结语

综上所述，区块链环境下搭建 PKI 是一种可行的尝试，它给我们提供了全新的保存和获取用户公钥的方式。相较于现有的 PKI 体系，它有力地解决了现有体系中用户必须信任某一第三方中心的窘境，并有效降低了用户 pk 信息受攻击的风险，具有极强的鲁棒性，能抵抗存在性伪造和信息篡改；同时上述实验表明，它在现实中的应用是可行的，且当系统节点足够大时，对误操和攻击具有一定的韧性。在区块链技术正蓬勃发展的现在，区块链环境下的 PKI 的出现，将会是该技术在电子非对称通讯网络上的又一次重要尝试。

附录

PKI 智能合约运行代码

```
pragma solidity ^0.4.18;

contract PKI {

    // name owner
    struct Owner {
        bytes32 id;
        string name;
    }

    struct Identity {
        address _owner;
        bytes32 id;
        string name;
        bool regitered;
        bool updated;
        bool revoked;
        bytes20 pkn;
        bytes20 pkf;
    }

    address private pki;
    address private user;
    mapping (address => Owner) private owner;
    Identity[] public identitys;
     string registerp = "regitser";
    string updatep = "update";
    string revokep = "revoke";

    event PkRegister(bytes32 id,string name,string optype, bytes20 pkn, bytes20
pkf);
    event PkUpdate(bytes32 id,string name,string optype, bytes20 pko, bytes20 pkw,
uint findaddress);
    event PkRevoke(bytes32 id,string name,string optype, bytes20 pkn, bytes20 pkf,
uint findaddress);

    //signation
    //input 'web3.eth.sign(key, m)'.in console for singation m return bytes
```

```
    //sha3(...) returns (bytes32)
    // ecrecover（hash，r， s， v）return address in 20bytes；
    //signed m is in hex type

    function decode(bytes32 m, bytes memory signedString) private pure
returns(address){
        bytes32 r = bytesToBytes32(slice(signedString, 0, 32));
        bytes32 s = bytesToBytes32(slice(signedString, 32, 32));
        byte    v = slice(signedString, 64, 1)[0];
        return ecrecoverDecode(m, r, s, v);
     }

    function slice(bytes memory data, uint start, uint len) private pure returns (bytes){
        bytes memory b = new bytes(len);
        for(uint i = 0; i < len; i++){
            b[i] = data[i + start];
        }
        return b;
    }

    function ecrecoverDecode(bytes32 m, bytes32 r, bytes32 s, bytes32 v1) private
pure returns(address) {
        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
          m = keccak256(prefix, m);
        uint8 v = uint8(v1) + 27;
          address addr = ecrecover(m, v, r, s);
          return addr;
    }

     function bytesToBytes32(bytes memory source) private pure returns(bytes32
result) {
        assembly {
            result := mload(add(source, 32))
        }
    }

    //
    function verifysign(bytes32 hm, bytes value, bytes20 pk) private pure
returns(bool){
        address pkad = address(uint(pk));
        if (pkad == (decode(hm, value))) {
             return true;
        }
        return false;
```

```
    }

    //bytes32+bytes20
    function addbytes52(bytes32 id, bytes20 pk) private pure returns(bytes) {
        bytes memory addresult;
        for (uint i =0; i < 52; i++) {
            if (i < 32) {
                addresult[i] = id[i];
            }
            else addresult[i] = pk[i-32];
        }
        return addresult;
    }

    //bytes32+bytes
    function addbytes(bytes32 id,bytes memory revoke) private pure returns(bytes) {
        bytes memory addresult;
        for (uint i =0; i < (32+revoke.length); i++) {
            if (i < 32) {
                addresult[i] = id[i];
            }
            else addresult[i] = revoke[i-32];
        }
        return addresult;
    }



    //遍历查询数组identitys中的元素，由id返回identity在数组中位置
    function Findaddress(bytes32 id) private view returns (uint){
        for (uint p = 1; p < identitys.length; p++) {
            if (identitys[p].id == id) {
                return p;
            }
        }
        return 0;
    }

    //注册时同时提交on、off钥对,valuex为pkx为钥对id的签名
    function Register(bytes32 id, string name, bytes20 pkn, bytes20 pkf, bytes
valuen, bytes valuef) public {
        require(0 == Findaddress(id) || identitys[uint(Findaddress(id))].revoked);
        if (verifysign(id, valuef, pkf) && verifysign(id, valuen, pkn)) {
            identitys.push(Identity(msg.sender, id, name, true, false, false, pkn,
```

```
pkf));
                string storage optype = registerp;
                emit PkRegister(id, name, optype, pkn, pkf);
            }
        }

        //更新密钥，密钥类型规定on为1，off为2；valueo为pko为钥对(id+pkw)的签名，valuew为pkw对id的签名
        function update(bytes32 id,    string name, int keytype, bytes20 pkw, bytes20 pko, bytes valuew, bytes valueo) public {
            require(0 != Findaddress(id));
            uint thisaddr = Findaddress(id);
            require(!identitys[thisaddr].revoked && pko == identitys[thisaddr].id &&pkw != pko);
            bytes memory id_pkw = addbytes52(id, pkw);
            bytes32 idpkw = keccak256(id_pkw);
            require(verifysign(idpkw , valueo, pko) && verifysign(id, valuew, pkw));
            if (keytype == 1) {
                identitys[thisaddr].pkn = pkw;
            }
            else if (keytype == 2) {
                identitys[thisaddr].pkf = pkw;
            }
            identitys[thisaddr].updated = true;
            string storage optype = updatep;
            emit PkUpdate( id, name, optype, pko, pkw, thisaddr);
        }

        //注销操作,密钥类型规定on/off都要；valuenr为pkn为钥对(id+revoke)的签名，valuefr为pkf对id+revoke的签名
        function revoke(bytes32 id,    string name, bytes valuenr, bytes valuefr) public {
            require(0 != Findaddress(id));
            uint thisaddr = Findaddress(id);
            require(!identitys[thisaddr].revoked );
            string memory revokestring = "revoke";
            bytes memory id_revoke = addbytes(id, bytes(revokestring));
            bytes32 idrevoke = keccak256(id_revoke);
            require(verifysign(idrevoke , valuenr, identitys[thisaddr].pkn) && verifysign(idrevoke, valuefr, identitys[thisaddr].pkf));
            identitys[thisaddr].revoked = true;
            string storage optype = revokep;
            emit PkRevoke(id, name, optype, identitys[thisaddr].pkn, identitys[thisaddr].pkf, thisaddr);
        }
```

```
    //查找操作
    function Lookup(bytes32 id) public view returns (bytes20 pk) {
        require(0 != Findaddress(id));
        uint thisaddr = Findaddress(id);
        require(!identitys[thisaddr].revoked );
        return identitys[thisaddr].pkn;
    }
}
```

参考文献：

[1] 何蒲, 于戈, 张岩峰, 鲍玉斌. 区块链技术与应用前瞻综述[J]. COMPUTER SCIENCE, 2017, 44(4): 1- 8.

[2] 刘德林.区块链智能合约技术在金融领域的研发应用现状、 问题及建议[J]. 海南经济, 2016, 335(10): 26-31

[3] Stiglitz Joseph E，Weiss Andrew．Credit Rationing in Market with Imperfect Information [J]．The America Economic Review，1981(3)

[4] NAil<AMOTO S Bitcoin>：a peer-to-peer electronic cash system[EB/OI]. https://bitcoin.org/bitcoin.pdf.

[5] Ethereum. Ethereum homestead document[EB/OL]. 2018.3[2018.5.12]. http://ethdoc.cn.

[6] Carter J, Wegman M. Universal classes of hash functions (Extended Abstract) [J]. Acm Symposium on Theory of Computing , 1977, 18(2):106-112.CCarter

[7] 朱岩, 甘国华, 邓迪, 等. 区块链关键技术中的安全性研究[J]. 信息安全研究, 2016, 2(12):1090-1093.

[8] HANCOCK M,VAIZEY E.Distributed ledger technology：beyond block chain[R].UK：Government Office for Science，2016

[9] 谢冬青. PKI 原理与技术[M]. second sersion. 清华大学:清华大学出版社, 2004 :116-127.

[10] 冯登国. Cryptography: Theory and Practice[M]. WaterLoo:Douglas R.Stinson, 2003.

[11] Fromknecht C, Velicanu D, Yakoubov S. A Decentralized Public Key Infrastructure with Identity Retention[D]. MIT, 2014.

[12] Stoica I, Morris R, Karger D, etal. Chord:A scalable peer-to-peer lookup service for internet applications [C]. //, Acm Conference of the Special Interest Group on Date Communication, 2001:149-160.

[13] Maymounkov P, David Mazi`eres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric[C]. //CCR 0093361, New York University: National Science Foundation, 2001.

[14] Wytse, Oortwijn. A Distributed Hash Table for Shared Memory[D]. University of Twente:Wytse Oortwijn(B), Tom van Dijk, and Jaco van de Pol, 2017. 1-10

[15] Zhuo C, Gang F, Yi L, etal. Improving Playback Quality of Peer--to--Peer Live Streaming Systems by Joint Scheduling and Distributed Hash Table Based Compensation[D]. ,Atlanta 30332，USA:College ofComputer Science and Engineering，Chongqing University ofTechnology, 2004.

[16] Ethereum. Solidity Documentation[M]. Release 0.4.23. Ethereum :Ethereum , 2018 :16-235.

[17] Kosba. Ahmed EMiller, AnmShi, Elaine,.Wen, Zikai and Papamanthou. Charalampos.The blockchain model of cryptography and privacy-preserving smart contracts[J]．IACR Cryptology ePrint Archive，2015：67

谢辞

本论文从课题的选取到项目的最终完成，都得到了导师王美琴老师的细心指导和不懈支持。在整个研究阶段，她精益求精的工作作风，诲人不倦的高尚师德，都教诲着我：不能浪费宝贵的时间，要及时学习让时光不能虚度；因此无论是每周的进展报告，还是最后对论文的调整，都对我课题的进展做出重要的监督和提升作用。同时还要感谢王继志老师，在他的亲切关怀和悉心指导下，我在研究方向出现偏差时得以及时纠正，并为我在进展中出现的许多问题的的解决提供了重要的帮助，在此谨向王美琴和王继志老师致以诚挚的谢意和崇高的敬意。感谢山东大学数学学院学院的老师对我的教育的精心培养和学习的细心指导，在此，我要向诸位老师深深地鞠上一躬。感谢给我带给参考文献的学者们，多谢他们给我的文献，使我在写论文的过程中有了参考的依据。也谢谢即将毕业的同学朋友们，在相互的鼓励和努力之下，一起走到现在。谢谢大家。

引用译文

文献一：< Decentralized Public Key Infrastructure with Identity Retention **>**

Public key infrastructures (PKIs) enable users to look up and verify one another's public keys based on identities. Current approaches to PKIs are vulnerable because they do not offer suffciently strong guarantees of identity retention; that is, they do not effectively prevent one user from registering a public key under another's already-registered identity. In this paper, we leverage the consistency guarantees provided by cryptocurrencies such as Bitcoin and Namecoin to build a PKI that ensures identity retention. Our system, called Certcoin, has no central authority and thus requires the use of secure distributed dictionary data structures to provide effcient support for key lookup.

In this paper, we describe Certcoin, a new completely decentralized PKI that leverages the consistency offered by blockchain-based cryptocurrencies such as Namecoin to provide a stronger identity retention guarantee than any of the PKIs used in practice. Instead of having to trust a third party as in the CA system or a small set of fellow users as in the PGP system, Certcoin only requires that users trust that the majority of other users are not malicious. There have been other works similarly leveraging blockchain systems, such as the work of Garman, Green and Miers on the topic of anonymous credentials. However, to our knowledge, the use of a blockchain for the instantiation of a full- edged PKI has never been fully explored. We detail how Certcoin can effciently support each PKI functionality. Because of the decentralized nature of this PKI, there is no single authority who can maintain a local dictionary data structure for effcient public key lookup. We therefore separate the functionality of verifying a known public key from that of looking up a new public key, and leverage secure distributed data structures to support each of them effciently. In particular, we use cryptographic accumulators to facilitate fast public key veriffcation, and distributed hash tables to facilitate fast public key lookup.

公钥基础设施（PKI）使用户能够根据身份查找并验证彼此的公钥。目前对

公钥基础设施的做法很脆弱，因为它们不能提供充分有力的身份保留保证;也就是说，它们不能有效地防止一个用户在另一个已经注册的身份下注册公钥。在本文中，我们利用加密货币（如比特币和 Namecoin）提供的一致性保证来构建可确保身份保留的 PKI。 我们的系统称为 Certcoin，没有中央管理机构，因此需要使用安全的分布式字典数据结构来为密钥查找提供高效的支持。

在本文中，我们描述了 Certcoin，一种新的完全分散的 PKI，它利用区块链加密货币（如 Namecoin）提供的一致性，提供比任何实践中使用的 PKI 更强的身份保留保证。 Certcoin 只需要用户相信大多数其他用户不是恶意的，而不必像 CA 系统中那样信任第三方或者像 PGP 系统中的一小部分同伴用户。还有一些其他的工作同样利用区块链系统，例如 Garman，Green 和 Miers 关于匿名凭证主题的工作。但是，就我们所知，使用区块链来实现全面的 PKI 实例从来没有得到充分的探索。我们详细介绍了 Certcoin 如何有效支持每个 PKI 功能。由于这种 PKI 的分散性，没有单一的权力机构可以维护本地字典数据结构以进行有效的公钥查找。因此，我们将验证已知公共密钥的功能与查找新公钥的功能分开，并利用安全的分布式数据结构来有效支持它们中的每一个。特别是，我们使用加密累加器来实现快速公钥检测，并使用分布式哈希表来实现快速公钥查找。

文献二：<Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications>

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents Chord, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis, simulations, and experiments show that Chord is scalable, with communication cost and the state maintained by each node scaling logarithmically with the number of Chord nodes.

At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. It uses consistent hashing [11, 13],

which has several good properties. With high probability the hash function balances load (all nodes receive roughly the same number of keys). Also with high probability, when an N^{th} node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location --- this is clearly the minimum necessary to maintain a balanced load.

Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few other nodes. In an N-node network, each node maintains information only about $O(logN)$ other nodes, and a lookup requires $O(logN)$ messages. Chord must update the routing information when a node joins or leaves the network; a join or leave requires $O(log^2N)$ messages

面对 peer-to-peer 应用程序的一个基本问题是有效地定位存储特定数据项的节点。 本文介绍了 Chord，这是一个解决这个问题的分布式查找协议。Chord 只支持一个操作：给定一个键，它将键映射到一个节点上。 通过将密钥与每个数据项相关联，并将密钥/数据项对存储在密钥映射到的节点上，可以在 Chord 之上轻松实现数据位置。Chord 可以在节点加入和离开系统时高效地进行调整，即使系统不断变化，也可以回答查询。 理论分析，仿真和实验结果表明，Chord 具有可扩展性，其通信成本和每个节点所维护的状态都以 Chord 节点的数量为对数进行扩展。

Chord 的核心是提供散列函数的快速分布式计算，将密钥映射到负责它们的节点。它使用一致的哈希[11,13]，它有几个好的属性。散列函数平衡负载的可能性很高（所有节点接收的密钥数量大致相同）。也很有可能，当一个N^{th}节点加入（或离开）网络，只有一个$O(1/N)$密钥的一小部分移动到不同的位置 - 这显然是维持平衡负载所需的最低限度.Chord 通过避免每个节点都知道其他每个节点的要求，改进了一致性哈希的可扩展性。 Chord 节点只需要少量有关其他节点的“路由”信息。由于这些信息是分布式的，所以一个节点通过与其他几个节点进行通信来解析散列函数。在一个 N—节点网络，每个节点只保存信

息O(logN)其他节点和需要查找O(logN)消息。Chord 必须在节点加入或离开网络时更新路由信息;加入或离开需要? O(log^2N)消息量。

文献三：〈A Distributed Hash Table for Shared Memory 〉

Distributed algorithms for graph searching require a highperformance CPU-efficient hash table that supports find-or-put. This operation either inserts data or indicates that it has already been added before. This paper focuses on the design and evaluation of such a hash table, targeting supercomputers. The latency of find-or-put is minimized by using one-sided RDMA operations. These operations are overlapped as much as possible to reduce waiting times for roundtrips. In contras to existing work, we use linear probing and argue that this requires less roundtrips. The hash table is implemented in UPC. A peak-throughput of 114.9 million op/s is reached on an Infiniband cluster. With a load-factor of 0.9, find-or-put can be performed in 4.5 μs on average. The hash table performance remains very high, even under high loads.

In this section the hash table structure and the design of find-or-put are discussed. We expect linear probing to require less roundtrips than both Cuckoo hashing and Hopscotch hashing, due to the absence of expensive relocation mechanisms. We also expect that minimising the number of roundtrips is key to increased performance, since the throughput of the hash table is directly limited by the throughput of the RDMA devices. This motivates the use of linear probing in the implementation of find-or-put. Unlike , we only use linear probing, since it reduces latency compared to quadratic probing, at the cost of possible clustering. We did not observe serious clustering issues, but if clustering occurs, quadratic probing can still be used, at the cost of slightly higher latencies. The latency of find-or-put depends on the waiting time for roundtrips to remote memory (which is also shown in Sect. 4). We aim to minimize the waiting times by overlapping roundtrips as much as possible, using asynchronous memory operations. Furthermore, the number of roundtrips required by find-or-put is linearly reduced by querying for chunks instead of individual buckets. We use constant values C to denote the chunk size and M to denote the maximum number of chunks that find-or-put takes into account. Figure 1

shows the design of find-or-put. Design considerations are given in the following sections.

用于图搜索的分布式算法需要支持查找或放入的高性能 CPU 高效哈希表。该操作要么插入数据要么表明它已经被添加过。本文着重于这种散列表的设计和评估，针对超级计算机。 通过使用单面 RDMA 操作，查找或放置的延迟被最小化。这些操作尽可能重叠以减少往返时间的等待时间。与现有工作相反，我们使用线性探测并认为这需要较少的往返。哈希表在 UPC 中实现 Infiniband 集群达到峰值吞吐量 1.149 亿 op / s。在 0.9 的负载因子下，平均可在 4.5 μ s 内执行查找或放置操作。即使在高负载下，哈希表性能仍然非常高。

对于散列表结构和 find-or-put 的设计。由于缺乏昂贵的重定位机制，我们预计线性探测所需的往返次数少于 Cuckoo 哈希和 Hapscotch 哈希。我们也期望将往返次数最小化是提高性能的关键，因为哈希表的吞吐量直接受到 RDMA 设备吞吐量的限制。这激励了在实施查找或放置中使用线性探测。与之不同的是，我们只使用线性探测，因为它可以减少与二次探测相比的延迟，代价是可能的聚类。我们没有观察到严重的聚类问题，但是如果发生聚类，仍可以使用二次探测，代价是略高的延迟。find-or-put 的等待时间取决于往返于远程存储器的等待时间（这也在第 4 节中显示）。我们的目标是通过使用异步存储器操作尽可能地重叠往返来尽量减少等待时间。此外，查找或放置所需的往返次数通过查询块而不是单个桶来线性减少。我们使用常量值 C 来表示块的大小，M 用来表示查找或放入的块的最大数目。图 1 显示了查找或放置的设计。设计注意事项在下面的章节中给出。